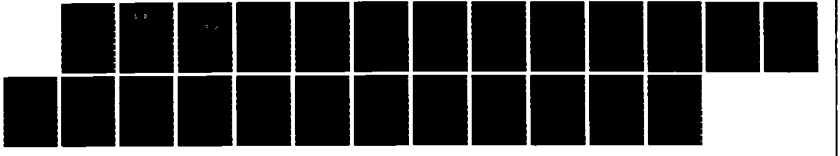
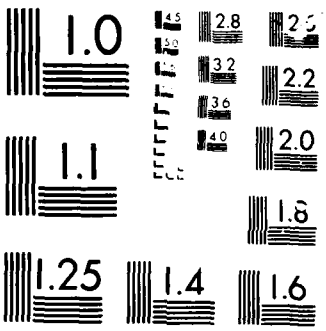
AD-A168 697 SYSTOLIC ARRAY SYNTHESIS: COMPUTABILITY AND TIME CONES (U) YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE J DELOSME ET AL. MAY 86 YALEU/DCSRR-474 1/1

UNCLASSIFIED N00014-82-K-0184 F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.2
2.0
1.8

(13)

Systolic Array Synthesis: Computability and Time Cones

Jean-Marc Delosme[1], Ilse C.F. Ipsen[2]

Research Report YALEU/DCS/RR-474
May 1986

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract. Many important algorithms in signal and image processing, speech and pattern recognition or matrix computations consist of coupled systems of recurrence equations. Systolic arrays are regular networks of tightly coupled simple processors with limited storage that provide cost-effective high-throughput implementations of many such algorithms. While there are some mathematical techniques for finding efficient schedules for uniform recurrence equations, there is no general theory for more general systems of recurrence equations. The first elements of such a theory are presented in this paper and constitute a significant step towards establishing a complete methodology that determines systolic array implementations for a very general class of coupled systems of recurrence equations; these implementations exhibit provably optimal computation time while satisfying various user-specified constraints.

Systolic Array Synthesis: Computability and Time Cones

Jean-Marc Delosme[1], Ilse C.F. Ipsen[2]

Research Report YALEU/DCS/RR-474
May 1986

[1] Department of Electrical Engineering, Yale University
[2] Department of Computer Science, Yale University

The work presented in this paper was supported by the Office of Naval Research under contracts N00014-82-K-0184, N00014-84-K-0092 and N00014-85-K-0461 and by the National Science Foundation under grant ECS-8314750.

1. Introduction

Systolic arrays, regular networks of tightly coupled simple processors with limited storage provide cost-effective high-throughput implementations of important algorithms in a variety of areas (signal and image processing, speech and pattern recognition, matrix computations). The simplest of these algorithms consist of a single recurrence equation while most of them are fairly complex systems of coupled recurrence equations. A recurrence equation specifies the computation of a variable at various index values as a function of other indexed variables. In this paper, we present the first steps towards a mathematical theory for the systematic implementation of coupled systems of recurrence equations on systolic arrays. The resulting arrays exhibit provably optimal computation time while satisfying various user-specified constraints. Our theory employs several concepts introduced in early works by Karp, Miller and Winograd [3, 4].

The starting-point for the systematic systolic implementation of recurrence equations was made with methods developed for the simplest class of equations. This is the class for which the difference between the indices of a used variable and the indices of the computed variable within a recurrence equation is constant (uniform recurrence equations [7] and regular iterative algorithms [8]). Recently, extensions have been attempted to cover the implementation of a larger class of equations: those containing variables with differing numbers of indices (e.g. [6]) or functions with large fan-ins, encountered in dynamic programming, for instance. These attempts, however, lack the rigour found in the afore-mentioned theories for uniform recurrence equations; they are expedients with a limited and not clearly delineated domain of application.

For more general systems of equations, a mathematical model must be established that captures the main properties of the recurrence equations under consideration, and mathematical methods consistent with the model should be used to perform a computability analysis and to determine efficient schedules. These recurrence equations are characterised by differences between the indices of computed and used variables that are affine, rather than constant, functions of the indices. The key uniformity property exploited in [4, 7, 8] is thus lost. Previous works tried to convert the affine equations to uniform ones, thus attempting to render the scheduling problem amenable to the techniques developed for uniform recurrence equations. We propose instead a theory that, for arbitrary affine functions of the indices, parallels at a fundamental level the work in [4]. When fully developed, it will permit the automatic determination of efficient implementations for the types of algorithms of concern in previous extensions as well as for some algorithms (of proven importance in signal processing) that do not fall into the categories covered by previous extensions.

We believe that the proper implementation of recurrence equations on systolic arrays (or any other parallel architecture, for that matter) necessitates a determination of the computability of the equations. The information gathered during the verification phase can then be profitably used in the scheduling and processor assignment phase.

In Section 2 we introduce the concept of computability of a variable: a variable is not computable if there exists an index value P so that u_P depends on itself, possibly through a number of intermediate variables. The renaming algorithm presented in Sections 3 and 4 identifies and extracts (i.e. gives a new name to) those 'harmless' instances of each variable that can never be responsible for a self-reference. The variables associated with the remaining, potentially harmful, instances form groups (the 'steps' of the algorithm) in which they mutually depend on each other. Hence, each variable u in a step gives rise to 'cyclic' dependences of the form: u_P depends on u_Q. A computability analysis then determines whether Q happens to be equal to P for some P.

A direct application of the definition of computability would lead to the exhaustive examination of possible self-dependences for all instances of a variable u in a step. The notion of dependence mapping (the difference between the indices of a used variable and the ones of the computed variable), introduced in Section 2, allows us to look at all instances of a variable at once instead of

looking at every instance in turn. It is sufficient, as shown in Sections 5 and 6, to consider compositions of dependence mappings around cycles instead of individual dependence mappings for the verification of computability. Moreover, for affine (or uniform) dependence mappings, computability tests can be performed by examining those properties of compositions of dependence mappings around cycles that are invariant with respect to affine (or uniform) similarity transformations – so as to keep the dependence mappings affine (or uniform). Necessary and sufficient conditions for the computability of uniform recurrence equations are given in Section 6.

In Sections 2 to 6 the computability analysis is shown to necessitate the decomposition of algorithms into steps. This same decomposition will also be used for scheduling the computations in the algorithm. The scheduling process first determines independently for each step *all* its possible schedules and then selects through an optimisation process *one* schedule for each step in order to obtain an efficient schedule and processor assignment for the whole algorithm. With regard to uniform recurrence equations, Section 7 introduces the notion of a 'time cone' that concisely characterises the totality of schedules for a step, and describes how to obtain a particularly simple type of schedule from the time cone. Although a step is computable it may not necessarily possess a time cone, the dependence mappings are then converted to affine ones in order to find an efficient schedule for the step. The concluding section gives a flavour of the scheduling of affine dependence mappings that we plan to investigate in depth as we further develop our theory.

2. Verification of Computability

The algorithms to be considered consist of coupled systems of recurrence equations. In a *recurrence equation* the computation of a variable at various index values is specified as a function of other indexed variables (by convention, a variable is computed only *once* for each index value, thus no overwriting is allowed).

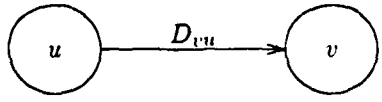
> As an example consider the algorithm that computes for a given symmetric Toeplitz matrix both its LDL^T factorisation and the LDL^T factorisation of its inverse.
>
> Toeplitz Factorisation Algorithm:

$$
\begin{aligned}
1 \le i \le n, &\quad r_{i,0} \equiv a_{i-1} \\
2 \le i \le n, &\quad s_{i,0} \equiv a_{i-1} \\
1 \le j \le n-1, &\quad \rho_j = s_{j+1,j-1}/r_{j,j-1} \\
&\quad j+1 \le i \le n, \quad r_{i,j} = r_{i-1,j-1} - \rho_j s_{i,j-1} \\
&\quad j+2 \le i \le n, \quad s_{i,j} = -\rho_j r_{i-1,j-1} + s_{i,j-1}
\end{aligned}
$$

$$
\begin{aligned}
&s_{1,0} = 1 \\
&1 \le j \le n-1, \quad 1 \le i \le j+1, \quad s_{i,j} = -\rho_j s_{j+2-i,j-1} + s_{i,j-1}
\end{aligned}
$$

Any scheduling procedure should a fortiori be able to determine whether a coupled system of recurrence equations is computable, specifically it should be able to verify that there is no instance where the computation of a variable at some index value depends on that variable at the same index value. A particular instance would be that a variable with index value P depends on another variable with index Q which in turn depends on the first one at P, thus forming a cycle through one intermediate variable. Verifying computability calls for the examination of all possible cycles, going through any number of intermediate variables. The solution to this problem represents a cornerstone in our scheduling procedure.

The verification process should exploit the structure of the computations and check that given coupled systems of recurrence equations do not exhibit any cycle – without direct examination of



$$D_{vu}: \quad P \mapsto Q$$
$$C_{vu} - R_{vu}$$

Figure 1: Illustration of Dependence Mapping as introduced in Definition 2.1.

the variables at *every* index value. In other words, the time to detect a cycle should be independent of the index range of the variables. To this end, we shall deal with *sets* of index points, called domains and ranges, and mappings that relate the two. Explicitly, one defines:

Definition 2.1. The mapping $Q = D_{vu}(P)$ associated with an equation $u_P = f(v_Q, \ldots)$ is called *dependence mapping* (the precedence of v over u in the subscripts indicates that v at point Q must be available *before* the computation of u at point P can commence). The *domain of computation* C_{vu} is the set of index points P for which the variable u is computed according to the equation $u_P = f(v_Q, \ldots)$, it is the *domain* of D_{vu}. The set of points Q at which variable v is needed in order to compute u_P for all $P \in C_{vu}$ is denoted by R_{vu} and is the *range* of D_{vu}. These notions are illustrated in Figure 1.

The dependence mapping $D_{vu}(P) = Q$ associated with a recurrence equation $u_P = f(v_Q, \ldots)$ represents the index value Q of v as a function of the index value P at which u is computed.

For the Toeplitz factorisation algorithm the dependence mappings are

$$\begin{aligned}
&D_{ar}(i,0) = i-1, \quad D_{as}(i,0) = i-1 \\
&D_{\rho r}(i,j) = j, \quad D_{rr}(i,j) = (i-1,j-1), \quad D_{sr}(i,j) = (i,j-1) \\
&D_{\rho s}(i,j) = j, \quad D_{rs}(i,j) = (i-1,j-1) \\
&D_{ss}^{(1)}(i,j) = (i,j-1), \quad D_{ss}^{(2)}(i,j) = (j+2-i,j-1) \\
&D_{r\rho}(j) = (j,j-1), \quad D_{s\rho}(j) = (j+1,j-1).
\end{aligned}$$

and the corresponding domains of computation are

$$\begin{aligned}
&C_{ar} = \{(i,0) : 1 \le i \le n\}, \quad C_{as} = \{(i,0) : 2 \le i \le n\} \\
&C_{\rho r} = C_{rr} = C_{sr} = \{(i,j) : j+1 \le i \le n,\ 1 \le j \le n-1\} \\
&C_{\rho s} = \{(i,j) : 1 \le i \le n,\ 1 \le j \le n-1\}, \quad C_{rs} = \{(i,j) : j+2 \le i \le n,\ 1 \le j \le n-1\} \\
&C_{ss}^{(1)} = \{(i,j) : 1 \le i \le n,\ 1 \le j \le n-1\}, \quad C_{ss}^{(2)} = \{(i,j) : 1 \le i \le j+1,\ 1 \le j \le n-1\} \\
&C_{r\rho} = C_{s\rho} = \{j : 1 \le j \le n-1\}.
\end{aligned}$$

The ranges may readily be determined from the dependence mappings and their domains.

3. Renaming

During the process of verifying the computability of a variable one realises that certain instances of this variable could not possibly result in a self-reference. The purpose of renaming is to identify and extract for each variable its 'harmless' instances so that the remaining instances may then be subjected to a verification process. This is accomplished by dividing up the domains of computation, and distinguishing the resulting subdivisions by giving the associated instances of the variable a

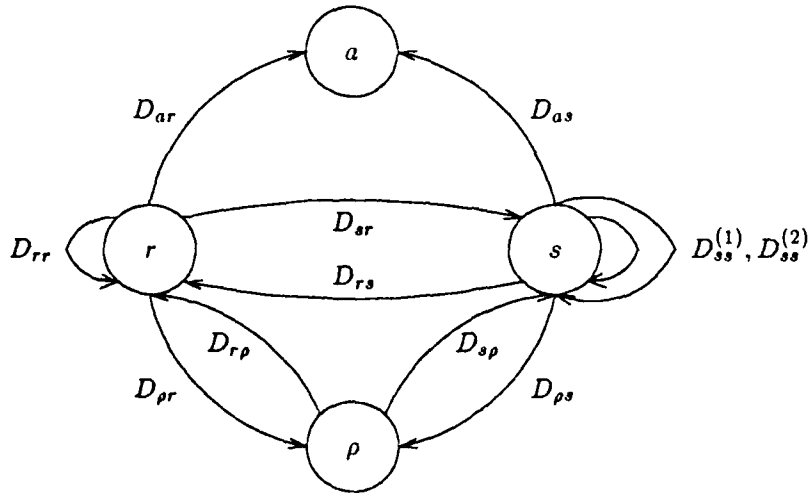


Figure 2: Dependence Graph for the Toeplitz Factorisation Algorithm.

new name. As a result each variable may be identified with exactly one domain of computation, and only certain variables need to be examined for computability.

At first, a directed *dependence graph* [4, 5] is constructed whose nodes correspond to the variables and whose arcs represent the dependences between variables. Figure 2 shows the dependence graph for the recurrence equations of the example algorithm. However, this graph captures only the dependence with regard to variable ***names*** but does not take into account ***index values***. For instance, one could infer from Figure 2 that the composition of dependence mappings

$$D_{sr} \circ D_{rs} \circ D_{ss}^{(2)}$$

may lead to a cycle while in fact one cannot perform this composition since the range of $D_{ss}^{(2)}$ and the domain of D_{rs} do not intersect. The computability analysis would be greatly simplified if *any* composition of dependence mappings associated with a cycle, i.e. a closed path, in the dependence graph were well-defined (these cycles should not be mistaken for the cycles encountered earlier, which are cycles in a larger graph where each node corresponds to a variable at a particular index point). Thus in each cycle of the dependence graph the range of the dependence mapping associated with an arc is required to have a non-empty intersection with the domain of the mapping associated with the next arc in the cycle. This may be ensured by properly assigning variable names to the left-hand sides of recurrence equations. Consequently, the first step in our scheduling procedure will perform a 'renaming' of the variables in the algorithm.

4. Renaming Procedure

1. *Construct the dependence graph of the algorithm.*
2. *Determine the strongly connected components of the dependence graph* via an efficient procedure such as the one in [9] (see also [4] and π-blocks in [5]). The dependence graph for the example contains only two strongly connected components: $\{r, s, \rho\}$ and the single node a. Since all dependence mappings involved in cycles incident on the same node must correspond to arcs within the same strongly connected component in the dependence graph, the renaming may be performed by examining each strongly connected component in turn. Thus the

remaining steps of the procedure are applied to every strongly connected component in the dependence graph.

3. *Find all 'well-defined' elementary cycles within a strongly connected component*, where a cycle is ***elementary*** if its arcs have all different initial endpoints and it is ***well-defined*** if the dependence mappings associated with its arcs can be composed according to the order imposed by the cycle. This is done by checking for each elementary cycle within the strongly connected component whether the range of each of its dependence mappings intersects the domain of the subsequent dependence mapping.

4. *Find those cycles which can be iterated arbitrarily many times*, in these 'iterative' cycles the number of possible iterations is unbounded as the cardinality of the domains of the variables involved in the cycle is increased. This is done by performing a test whose description follows for each (well-defined) cycle.

   Let γ be the cycle under consideration and let u be a node in γ; denote by $D_{\gamma u} : C_{\gamma u} \rightarrow R_{\gamma u}$ the composition of dependence mappings around the cycle starting at u, where $C_{\gamma u}$ is the domain of the first dependence mapping in the cycle γ when starting at u, and $R_{\gamma u}$ intersects $C_{\gamma u}$ since the cycle is well-defined. The lth iterate of $D_{\gamma u}$, i.e. the composition of $D_{\gamma u}$ with itself repeated l times, is denoted by $D^l_{\gamma u} : C_{\gamma u} \rightarrow R^{(l)}_{\gamma u}$. The range of $D^l_{\gamma u}$ may be determined through the recurrence

$$\begin{aligned} C^{(1)}_{\gamma u} &\equiv C_{\gamma u}, \quad R^{(1)}_{\gamma u} \equiv R_{\gamma u} \\ 2 \le k \le l, \quad C^{(k)}_{\gamma u} &= R^{(k-1)}_{\gamma u} \cap C_{\gamma u} \\ R^{(k)}_{\gamma u} &= D_{\gamma u}(C^{(k)}_{\gamma u}). \end{aligned}$$

   The cycle is *iterative* if and only if $|C^{(l+1)}_{\gamma u}|$ may be made non-zero for any l by selecting the problem size, hence $|C_{\gamma u}|$, sufficiently large. This condition is equivalent to being able to make the ratio $|C^{(l+1)}_{\gamma u}|/|C_{\gamma u}|$ non-zero. The ratio $|C^{(l+1)}_{\gamma u}|/|C_{\gamma u}|$ characterises the 'shrinkage' of the computation domain after l iterations of $D_{\gamma u}$ and is just the product of the 'incremental' shrinkages

$$s^{(k)}_{\gamma u} \equiv \frac{|C^{(k+1)}_{\gamma u}|}{|C^{(k)}_{\gamma u}|}, \qquad k = 1, 2, \ldots, l.$$

   Therefore the cycle γ is iterative if and only if all $s^{(k)}_{\gamma u}$, $1 \le k \le l$, may be made non-zero. Note that one need only construct the sequence of the ratios $s^{(k)}_{\gamma u}$ for a single node u along γ in order to determine if γ is iterative.

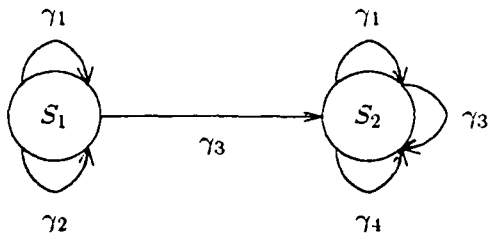


Figure 3: Cycle Composition Graph for Variable s in Toeplitz Factorisation Algorithm.

For illustration consider the cycle γ associated with the dependence mappings $D_{r\rho}$ and $D_{\rho r}$, the compositions along the cycle are

$$D_{\gamma r}(i,j) = D_{r\rho} \circ D_{\rho r}(i,j) = (j, j-1),$$
$$D_{\gamma\rho}(j) = D_{\rho r} \circ D_{r\rho}(j) = j - 1;$$

we also have

$$C_{\gamma r} = C_{\rho r}, \qquad C_{\gamma\rho} = C_{r\rho},$$

and it is easily seen that

$$s_{\gamma r}^{(1)} = \frac{2(n-2)}{n(n-1)}, \qquad s_{\gamma r}^{(l)} = \frac{n-1-l}{n-l} \qquad l = 2, 3, \ldots$$
$$s_{\gamma\rho}^{(l)} = \frac{n-1-l}{n-l} \qquad l = 1, 2, \ldots.$$

The cycle is iterative since $s_{\gamma r}^{(l)}$ (and $s_{\gamma\rho}^{(l)}$) is not identically 0 for any l when viewed as a function of the domain size parameter n.

5. *Partition the domains of computation for each variable.* For each node u, determine those pairs of well-defined elementary cycles incident on u that can be composed with each other as follows.

   Consider all (well-defined) cycles γ_k incident on u, and partition the set $\bigcup_k C_{\gamma_k u}$ into the subsets C_{ui} induced by the intersections and differences of the domains $C_{\gamma_k u}$.

   Consider node s in the strongly connected component $\{r, s, \rho\}$. Four elementary cycles γ_k, $1 \le k \le 4$, are incident on that node, corresponding to $D_{ss}^{(1)}$, $D_{ss}^{(2)}$, $D_{s\rho} \circ D_{\rho s}$, and $D_{sr} \circ D_{rs}$, respectively. The intersections and differences of the domains $C_{\gamma_k s}$ induce the partition of $\bigcup_k C_{\gamma_k s}$ into $C_{s1} \cup C_{s2}$ where

$$C_{s1} = \{(i,j) : 1 \le i \le j+1,\ 1 \le j \le n-1\},$$
$$C_{s2} = \{(i,j) : j+2 \le i \le n,\ 1 \le j \le n-1\}.$$

   Construct the directed graph whose nodes u_i correspond to the subsets C_{ui}. There is an arc, labelled γ_k, from node C_{ui} to node C_{uj} if there exists $P \in C_{ui}$ and $Q \in C_{uj}$ such that $Q = D_{\gamma_k u}(P)$; in other words, there is an arc from C_{ui} to C_{uj} if $D_{\gamma_k u}(C_{\gamma_k u} \cap C_{ui}) \cap C_{uj} \neq \emptyset$.

Continuing with the example, from

$$C_{\gamma_1 s} = C_{s1} \cup C_{s2}, \quad C_{\gamma_2 s} = C_{s1}$$
$$C_{\gamma_3 s} = C_{s1} \cup C_{s2}, \quad C_{\gamma_4 s} = C_{s2}$$

we have

$$\begin{aligned}
&C_{\gamma_1 s} \cap C_{s1} = C_{s1}, \quad D_{\gamma_1 s}(C_{s1}) \cap C_{s1} \neq \emptyset, \quad D_{\gamma_1 s}(C_{s1}) \cap C_{s2} = \emptyset \\
&C_{\gamma_1 s} \cap C_{s2} = C_{s2}, \quad D_{\gamma_1 s}(C_{s2}) \cap C_{s1} = \emptyset, \quad D_{\gamma_1 s}(C_{s2}) \cap C_{s2} \neq \emptyset \\
&C_{\gamma_2 s} \cap C_{s1} = C_{s1}, \quad D_{\gamma_2 s}(C_{s1}) \cap C_{s1} \neq \emptyset, \quad D_{\gamma_2 s}(C_{s1}) \cap C_{s2} = \emptyset \\
&C_{\gamma_2 s} \cap C_{s2} = \emptyset \\
&C_{\gamma_3 s} \cap C_{s1} = C_{s1}, \quad D_{\gamma_3 s}(C_{s1}) \cap C_{s1} = \emptyset, \quad D_{\gamma_3 s}(C_{s1}) \cap C_{s2} \neq \emptyset \\
&C_{\gamma_3 s} \cap C_{s2} = C_{s2}, \quad D_{\gamma_3 s}(C_{s2}) \cap C_{s1} = \emptyset, \quad D_{\gamma_3 s}(C_{s2}) \cap C_{s2} \neq \emptyset \\
&C_{\gamma_4 s} \cap C_{s1} = \emptyset \\
&C_{\gamma_4 s} \cap C_{s2} = C_{s2}, \quad D_{\gamma_4 s}(C_{s2}) \cap C_{s1} = \emptyset, \quad D_{\gamma_4 s}(C_{s2}) \cap C_{s2} \neq \emptyset.
\end{aligned}$$

The associated graph is displayed in Figure 3.

In general, find the partition of $\bigcup_k C_{\gamma_k u}$ induced by the strongly connected components of this graph. The elements in the partition are thus unions of subsets C_{ui}. This partition is called the 'composition-induced' partition of u. (In the reduced graph obtained from the condensation of each strongly connected component into a single node, the cycles γ_k incident on the same node may be composed in any order; this is the property we were looking for.)

> The graph in Figure 3 has two strongly connected components, s_1 and s_2. The analysis is even simpler for the two other variables r and ρ. Since all the $C_{\gamma_k r}$ are identical, the graph for r has only one node hence just one strongly connected component; the same is true for ρ.

6. *Explicitly rename instances of variables in the strongly connected component so that the set of well-defined cycles incident on the same variable is closed under composition*, i.e. all elementary cycles incident on the same variable may be iterated an arbitrarily large number of times and composed in arbitrary order.

   Renaming can be performed *independently* for each variable since it is based exclusively on the analysis of the elementary cycles incident on that variable, independent of the names of the nodes that these cycles traverse. For each variable u, it amounts to constructing a partition of $\bigcup_k C_{\gamma_k u}$ into nonoverlapping subsets and to giving distinct names to the instances of u whose indices belong to distinct subsets.

   In the actual process of renaming two issues have to be taken care of. Non-iterative cycles γ_k should be 'unrolled': this means that a different name is given to the instances of u whose indices belong to the subsets $C^{(i)}_{\gamma_k u} - C^{(i+1)}_{\gamma_k u}$, $1 \leq i \leq l$, where l is the maximum number of iterations of cycle γ_k (i.e. $C^{(l)}_{\gamma_k u} \neq \emptyset$ and $C^{(l+1)}_{\gamma_k u} = \emptyset$). The composition of distinct elementary cycles incident on the same variable should be well-defined: this means that distinct names are given to the instances of u according to the composition-induced partition of u.

   Since an instance u_P may depend on other instances of u through both iterative and non-iterative cycles, the two sources of renaming must be combined. This is simply done by renaming according to the composition-induced partition of $\bigcup_k C_{\gamma_k u}$ as well as the partitions of the $C_{\gamma_k u}$ associated with the non-iterative cycles γ_k into $\bigcup_i (C^{(i)}_{\gamma_k u} - C^{(i+1)}_{\gamma_k u})$.

No well-defined elementary cycle within the strongly connected component $\{r, s, \rho\}$ is non-iterative. Moreover the union of the domains associated with variable s is the only one to have a non-trivial composition-induced partition. Thus only the instances of s corresponding to the sets C_{s1} and C_{s2} will be given different names; we chose to use y for the index values in C_{s1} and to keep s for C_{s2}.

7. *Update dependence mappings and initialisations.*

To motivate this kind of post-processing consider once more the example. After renaming the last recurrence equation in the Toeplitz factorisation algorithm

$$1 \leq j \leq n-1, \quad 1 \leq i \leq j+1, \quad s_{i,j} = -\rho_j s_{j+2-i,j-1} + s_{i,j-1}$$

becomes

$$1 \leq j \leq n-1, \quad 1 \leq i \leq j+1, \quad y_{i,j} = -\rho_j y_{j+2-i,j-1} + y_{i,j-1}.$$

Observe, however, that $y_{1,0}$ is not defined since the renaming procedure left the assignment $s_{1,0} = 1$ unchanged. Clearly the instance $s_{1,0}$ must be converted to $y_{1,0}$.

Consequently, as renaming is performed, the dependence mappings must also be updated. Each dependence mapping D_{vu} is fully characterised by the mapping 'function' and its domain. If y is one of the new names for u then the dependence *functions* of y on other variables are just a subset of the dependence functions of u on other variables. While the functions remain the same, the new and old mappings differ in their domains (hence also ranges). The according, simple updating of domains is performed at this point for all variables in the algorithm.

For each renamed variable y consider the elementary cycles γ_k incident on that variable. The functions $D_{\gamma_k y}$ form a subset of the functions $D_{\gamma_k u}$ and, by construction, the domains of the mappings $D_{\gamma_k y}$ are all identical and equal to, say C_y. The instances of the old variable u whose indices belong to $\bigcup_k D_{\gamma_k}(C_y)$ and are outside $\bigcup_k C_{\gamma_k u}$ are renamed y. It may happen that the same instance of u is renamed more than once but this does not pose a problem as it corresponds only to additional assignments in the renamed algorithm (for example u could be renamed into y and z so that the assignment $u_{1,0} = 1$ would be replaced by $y_{1,0} = 1$ and $z_{1,0} = 1$).

Upon completion of this last renaming operation the renamed algorithm may be written in full, its dependence graph constructed and strongly connected components determined for further use. The new dependence graph will typically have more strongly connected components than the original dependence graph if some renaming has effectively occured. The portions of the algorithm corresponding to the strongly connected components in that graph are called the 'steps' of the algorithm.

To sum up, renaming ensures that the mappings corresponding to the elementary cycles within a step are all well-defined, can be iterated arbitrarily many times and composed in arbitrary order.

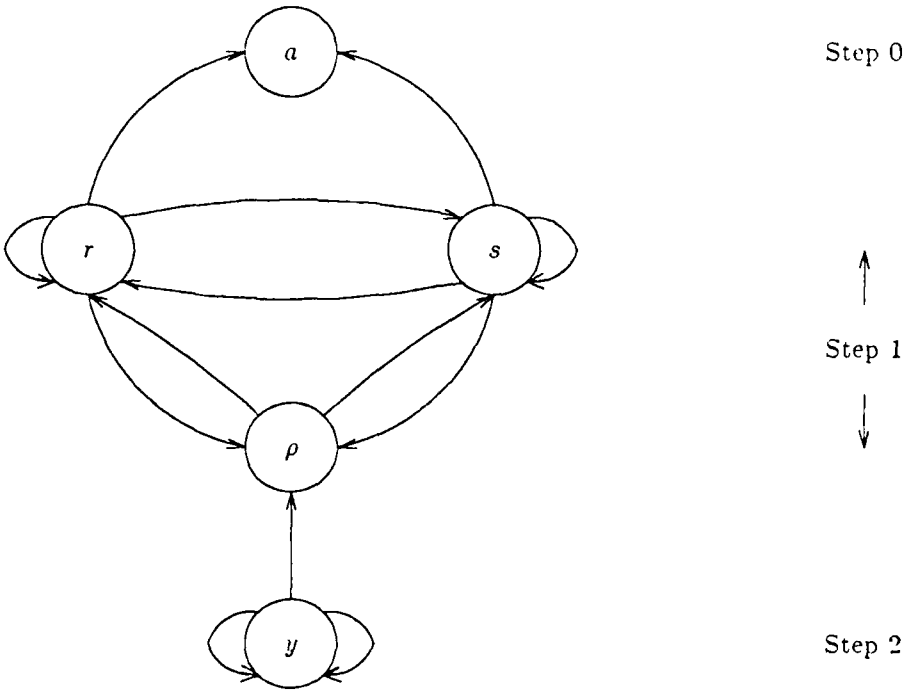


Figure 4: Dependence Graph for the Toeplitz Factorisation Algorithm after Renaming.

The new version of the Toeplitz factorisation algorithm after renaming follows.
Toeplitz Factorisation Algorithm:

$$
\begin{aligned}
1 \le i \le n, &\quad r_{i,0} \equiv a_{i-1} \\
2 \le i \le n, &\quad s_{i,0} \equiv a_{i-1} \\
1 \le j \le n-1, &\quad \rho_j = s_{j+1,j-1}/r_{j,j-1} \\
&\quad j+1 \le i \le n, \quad r_{i,j} = r_{i-1,j-1} - \rho_j s_{i,j-1} \\
&\quad j+2 \le i \le n, \quad s_{i,j} = -\rho_j r_{i-1,j-1} + s_{i,j-1}
\end{aligned}
$$

$$
\begin{aligned}
& y_{1,0} = 1 \\
& 1 \le j \le n-1, \quad 1 \le i \le j-1, \quad y_{i,j} = -\rho_j y_{i+2-i,j-1} + y_{i,j-1}
\end{aligned}
$$

The associated dependence graph is given in Figure 4.

5. Affine and Uniform Dependence Mappings

As a result of the renaming process, mappings associated with cycles in a step are closed under composition, and verifying the computability of a step amounts to an analysis of compositions of its cycles. The key observation of the analysis is based on the fact that self-dependence is invariant under a change of index: an instance $a_{\vec{P}}$ depends on itself if and only if $a_{\vec{P}}$ depends on itself, where $\vec{P} = F(\vec{P})$ and F is bijective. Thus computability is invariant with respect to changes of the indices of the variables, and it should therefore be verified by examining only those properties of mappings

associated with the cycles that are invariant (with respect to changes of indices).

To characterise these invariants, consider the effect of change of index transformations applied to every variable u

$$F_u : C_u \rightarrow \tilde{C}_u$$

where C_u is the domain of computation for variable u. Under these changes of indices a dependence mapping D_{vu} becomes

$$\tilde{D}_{vu} = F_v \circ D_{vu} \circ F_u^{-1},$$

so that the composition of dependence mappings $D_{\gamma u}$ associated with a cycle incident on a variable u undergoes a similarity transformation:

$$\tilde{D}_{\gamma u} = F_u \circ D_{\gamma u} \circ F_u^{-1}.$$

Consequently, computability is determined from those properties of the mappings associated with cycles that are invariant with respect to *similarity* transformations.

To carry the analysis further we shall now restrict the study to *affine* dependence mappings, that is, mappings of the form $Q = D(P)$, where $D(P) = DP + d$ is an affine transformation. Note that a non-linear dependence mapping may turn out to be an affine dependence mapping in disguise and, as such, can be transformed into an affine dependence mapping.

> The dependence mapping for the variable u in the recursion
>
> $$u_{(i-1)^2+1} = f(u_i), \qquad i \in \{3, 5, 17, \ldots, 2^{2^n}+1\}, \qquad u_3 \text{ given},$$
>
> where the index i runs over the set of the $n+1$ first Fermat numbers, is
>
> $$D_{uu}((i-1)^2+1) = i \qquad \forall i \in C_u = \{5, 17, \ldots, 2^{2^n}+1\}.$$
>
> Although D_{uu} is not an affine transformation, it can be expressed in the form
>
> $$D_{uu} = F_u^{-1} \circ \tilde{D}_{uu} \circ F_u, \qquad \text{where } F_u(i) = \log_2 \log_2 (i-1), \quad \tilde{D}_{uu}(j) = j - 1,$$
>
> and $\tilde{D}$ is an affine function. Thus, if the index set C_u is transformed via F_u then the resulting dependence mapping $\tilde{D}_{uu}$ is affine. Clearly, the transformation F_u is just a change of index and it does not change the computations of the recursion.

This situation should be recognised when it occurs since computability of affine dependences seems to be more easily verified than that of arbitrary dependences and, looking farther ahead, advantages will also accrue with regard to scheduling and processor assignment. Arbitrary non-linear dependence mappings, however, are not likely to be convertible to affine ones through change of index transformations F_u, as evidenced by the FFT algorithms. As already mentioned, all dependence mappings encountered from this point on will be affine. For affine dependence mappings, computability tests are performed by examining properties of cycles that are invariant with respect to *affine* similarity transformations F_u: the F_u should be affine in order to keep the dependence mappings affine.

A *uniform* dependence mapping is an affine dependence mapping of the form $D(P) = IP - d$, where I is the identity matrix. Thus a uniform dependence mapping is just a translation and the 'dependence vector' $D(P) - P$ is equal to d, independent of the point P in the domain. Recurrence equations for which all dependence mappings are uniform are called *uniform recurrence*

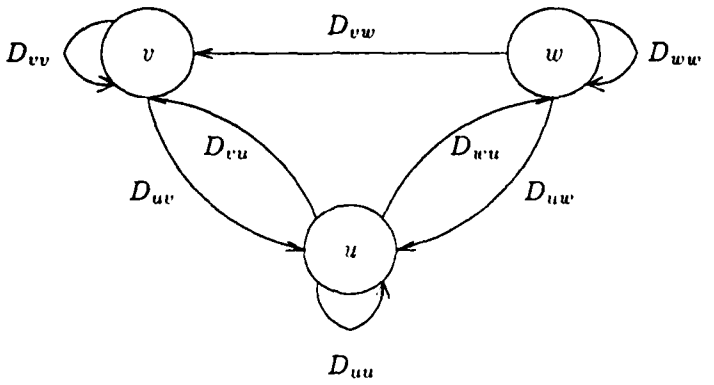


Figure 5: Dependence Graph for the Two-Dimensional Filter Example.

equations [4]. To test the computability of such algorithms one should consider properties of cycles that are invariant under *translations* F_* (in order to keep the dependence mappings uniform). The composition of uniform dependence mappings associated with a cycle is $D_{\gamma u}(P) = IP + d_{\gamma u}$ where $d_{\gamma u}$ is just the sum of the individual translation vectors d_{vu}; the vector $d_{\gamma u}$ clearly depends only on the cycle γ and not on the variable u at which the cycle is started, hence one can write $D_{\gamma u}(P) = IP + d_\gamma$. Under index changes of the form $F_*(P) \equiv IP + d_*$ a uniform dependence mapping $D_{vu}(P) \equiv IP + d_{vu}$ becomes

$$\check{D}_{vu}(P) \equiv IP + \check{d}_{vu}, \qquad \text{where } \check{d}_{vu} \equiv d_{vu} + (d_v - d_u),$$

and the dependence mapping $D_{\gamma u}$ around a cycle is left invariant. The translations d_γ around the elementary cycles within the dependence graph represent the invariants to be examined for computability verification.

6. Computability of Uniform Recurrence Equations

A direct application of the definition of computability would lead to the exhaustive examination of all self-dependences of a variable u in a step. The notion of dependence mapping allows us to look at all instances of a variable at once instead of looking at every instance in turn. We also know from before that it suffices to consider cycles instead of individual dependence mappings for the verification of computability. Moreover the renaming process ensures that any composition of cycles in a step is well-defined. Consequently, instead of exhaustively examining all self-dependences we consider arbitrary compositions of elementary cycles thus exploitating the property that the mappings are *uniform*.

Arbitrary compositions of elementary cycles γ_k within the step result in an overall translation

$$\sum_k \lambda_k d_{\gamma_k}, \qquad \lambda_k \geq 0.$$

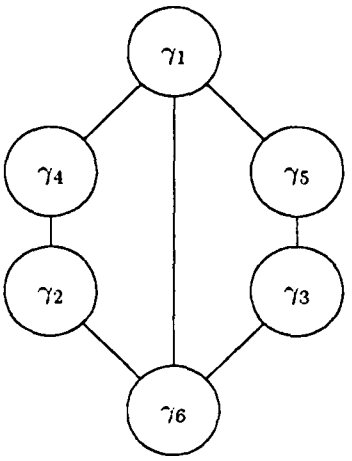


Figure 6: Cycle Composition Graph for the Two-Dimensional Filter Example.

where λ_k represents the number of times cycle γ_k is traversed. If a linear combination with positive coefficients is to correspond to a cycle through the dependence graph the subgraph formed by the nodes γ_k associated with the non-zero coefficients must constitute a connected component in the graph Γ that is defined as follows: the nodes of Γ are the elementary cycles in the step, and two nodes are connected by an edge if the corresponding cycles share a variable.

Two-Dimensional Filter Example:
In this example from [8] the filtered image $w_{n+1,j,k}$ is computed from the given image w_{1jk} according to the equations

$$\begin{aligned}
1 \le i \le n, \quad & u_{i00} = u_{i,-1,0} = u_{i,0,-1} \equiv 0 \\
1 \le j \le m,\ 1 \le k \le m, \quad & v_{njk} \equiv 0, \quad w_{1jk} \text{ given} \\
1 \le i \le n,\ 1 \le j \le m,\ 1 \le k \le m, \quad & u_{i,j+1,k+1} = f(u_{ijk}, v_{ijk}, w_{ijk}) \\
& v_{i-1,j,k} = g(u_{ijk}, v_{ijk}) \\
& w_{i+1,j,k} = h(u_{ijk}, v_{ijk}, w_{ijk})
\end{aligned}$$

The corresponding dependence graph is shown in Figure 5 and the translations in the dependence mappings are readily identified from the equations

$$\begin{bmatrix} d_{uu} & d_{vv} & d_{ww} & d_{uv} & d_{vu} & d_{uw} & d_{wu} & d_{vw} \\ 0 & 1 & -1 & 1 & 0 & -1 & 0 & -1 \\ -1 & 0 & 0 & 0 & -1 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 & -1 & 0 & -1 & 0 \end{bmatrix}.$$

There are six elementary cycles

$$\gamma_1 \equiv \{D_{uu}\}, \quad \gamma_2 \equiv \{D_{vv}\}, \quad \gamma_3 \equiv \{D_{ww}\}$$
$$\gamma_4 \equiv \{D_{uv}, D_{vu}\}, \quad \gamma_5 \equiv \{D_{uw}, D_{wu}\}, \quad \gamma_6 \equiv \{D_{wu}, D_{uv}, D_{vw}\}$$

whose translation vectors are easily seen to be

$$\begin{bmatrix} d_{\gamma_1} & d_{\gamma_2} & d_{\gamma_3} & d_{\gamma_4} & d_{\gamma_5} & d_{\gamma_6} \\ 0 & 1 & -1 & 1 & -1 & 0 \\ -1 & 0 & 0 & -1 & -1 & -1 \\ -1 & 0 & 0 & -1 & -1 & -1 \end{bmatrix}.$$

Figure 6 contains the graph Γ that indicates cycles with common variables.

A variable u is *not* computable if the following three conditions are satisfied:

1. The equation $\sum_k \lambda_k d_{\gamma_k} = 0$ has a non-trivial positive solution $\{\lambda_k\}$.
2. At least one of the cycles γ_k for which $\lambda_k \neq 0$ traverses u.
3. The subgraph in Γ formed by the nodes corresponding to $\lambda_k \neq 0$ is a connected component (this condition ensures that one can go from one cycle to another).

In general, a step is computable if all its variables are computable, i.e. only conditions 1 and 3 must be satisfied, omitting condition 2. Thus testing computability amounts to determining all the non-trivial positive solutions to

$$\sum_k \lambda_k d_{\gamma_k} = 0$$

and examining for each such solution the connectedness of the subgraph of Γ that is obtained by deleting the nodes corresponding to $\lambda_k = 0$.

The second and third components of the vectors d_{γ_k} reveal that any *positive* linear combination $\sum_k \lambda_k d_{\gamma_k} = 0$ must satisfy $\lambda_1 = \lambda_4 = \lambda_5 = \lambda_6 = 0$. Coefficients of the form $\lambda_2 = \lambda_3 > 0$ solve the remaining system $\lambda_2 d_{\gamma_2} + \lambda_3 d_{\gamma_3} = 0$. However, the nodes in Γ, γ_2 and γ_3, that are associated with λ_2 and λ_3, are not connected. Consequently, the algorithm is computable.

7. Scheduling of Uniform Recurrence Equations

It is clear from the previous sections that a computability analysis necessitates the decomposition of algorithms into steps. This same decomposition will also be used for scheduling the computations in the algorithm. The scheduling process first determines independently for each step *all* its possible schedules and then selects through an optimisation process *one* schedule for each step in order to obtain an efficient schedule and processor assignment for the whole algorithm. In this section we only describe the information characterising the schedules for an individual step and the way this information is derived. The global scheduling and processor assignment problem is presented and its solution discussed with the help of an example in [1, 2].

There is a fair amount of freedom in selecting the indices for the variables in the various recurrence equations. One can for instance choose not to have any relationship among the indices in each equation.

The recurrence equations in the 2-D filter example could be rewritten

$$1 \le i \le n,\ 1 \le j \le m,\ 1 \le k \le m,\ u_{i,j+1,k+1} = f(u_{ijk},\ v_{i'j'k'},\ w_{i''j''k''})$$
$$1 \le i' \le n,\ 1 \le j' \le m,\ 1 \le k' \le m,\ v_{i'-1,j',k'} = g(u_{ijk},\ v_{i'j'k'})$$
$$1 \le i'' \le n,\ 1 \le j'' \le m,\ 1 \le k'' \le m,\ w_{i''+1,j'',k''} = h(u_{ijk},\ v_{i'j'k'},\ w_{i''j''k''})$$

There is no relationship between $P_u = (i, j, k)$, $P_v = (i', j', k')$ and $P_w = (i'', j'', k'')$.

It is also possible to impose some relationship among the indices, for instance $P_u \equiv P_v \equiv P_w$ in the above example results in the original version of the algorithm. Once one decides to relate the indices of all variables in a step many choices are possible. All these choices may be obtained by replacing the index P_u of each variable u in a step by $F_u^{-1}(P)$, the bijections F_u being arbitrary and the index P being the same for all variables in the step.

A *schedule* for each step is defined through a suitable set of such transformations F_u and a vector τ, called 'time' vector, in such a way that an instance of a variable u with index P_u is computed at time $\tau^T P$ where $P = F_u(P_u)$. Of course not all transformations F_u and vectors τ define a schedule; they must also satisfy $\tau^T Q < \tau^T P$ whenever a variable at point Q is required for the computation of a variable at point P. The set $C \equiv \bigcup F_u(C_u)$ may be viewed as the computation domain for the whole step; in contrast to the domain C_u for a single variable more than one computation may be associated with a point P in C. Also observe that different choices of τ may result in schedules with widely differing degrees of parallelism.

For a cubical domain with $|C| = n^q$ the schedules

$$\tau = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \\ 1 \end{bmatrix}, \quad \tau = \begin{bmatrix} n \\ 1 \\ \vdots \\ 1 \\ 1 \end{bmatrix}, \quad \tau = \begin{bmatrix} n^2 \\ n \\ \vdots \\ 1 \\ 1 \end{bmatrix}, \quad \text{and} \quad \tau = \begin{bmatrix} n^{q-1} \\ n^{q-2} \\ \vdots \\ n \\ 1 \end{bmatrix}$$

result in respective computation times qn, n^2, n^3 and n^q up to lower order terms.

More specific schedules can be defined by requiring the transformations F_u to belong to a certain class. When the transformations F_u are restricted to be affine, i.e. $F_u(P_u) = F_u P_u + d_u$, the schedule for the uniform recurrence equations is called *affine*. A *translational* schedule is obtained if, in addition, $F_u \equiv I$ for all variables u.

We shall now characterise all *translational* schedules for a step that consists of uniform recurrence equations. Application of the transformations F_u results in $P = F_u(P_u) \equiv P_u + d_u$ and each dependence mapping $Q_v = D_{vu}(P_u) \equiv P_u + d_{vu}$ becomes $Q = P + \tilde{d}_{vu}$ where $\tilde{d}_{vu} = d_v + d_{vu} - d_u$. The set of translation vectors d_u and a time vector τ fully define a translational schedule subject to the condition that $\tau^T Q < \tau^T P$ whenever a variable at point Q is required for the computation of a variable at point P. This condition is easily seen to be equivalent to the condition that $\tau^T \tilde{d}_{vu} < 0$ for all the dependence mappings in the strongly connected component. The notion of 'time cone' provides a characterisation of all translational schedules, this characterisation can be used for the fast determination of translational schedules that have certain desired properties and for the solution of the global scheduling problem.

For each elementary cycle γ_k in the strongly connected component let ν_k be the number of arcs in the cycle and d_{γ_k} be the translation vector associated with the cycle. The *time cone* of the strongly connected component is the set of vectors τ satisfying $\tau^T d_{\gamma_k} \le -\nu_k g$ for *all* elementary cycles γ_k in the component, where g is the greatest common divisor of the components of τ.

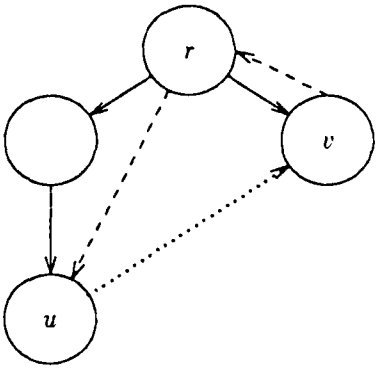


Figure 7: Tree T from the Single-Source Maximum-Cost Algorithm in Theorem 7.1.

The following theorem provides the main justification for the above definition.

Theorem 7.1. *For each τ in the time cone one can determine translation vectors d_u such that for every arc (u, v) in the strongly connected component $\tau^T \tilde{d}_{vu} \leq -g$, where $\tilde{d}_{vu} = d_{vu} + d_v - d_u$.*

Proof. Let τ be a vector in the time cone. With every arc (u, v) in the strongly connected component G (in particular each arc in a set of multiple arcs) associate the cost $c_{vu} = \tau^T d_{vu} + g$. Since τ belongs to the time cone, the sum of c_{vu} along a cycle is at most zero.

Select a starting node r and construct the tree T resulting from the single-source maximum-cost algorithm (this is just a single-source shortest-path algorithm with cost equal to $-c_{vu}$ for each arc (u, v)). Then, starting from the root r, assign to each node u an integer c_u such that for every arc (u, v) in T the new cost is $\tilde{c}_{vu} = c_{vu} + c_v - c_u = 0$ ($c_r \equiv 0$). We claim that $\tilde{c}_{vu} \leq 0$ for all arcs (u, v) in G.

Observe that the arcs in $G - T$ can be grouped into three categories:

- forward arcs (from a node to a descendent in T, solid line in Figure 7)
- backward arcs (from a node to an ascendant in T, dashed line in Figure 7)
- cross arcs (the remaining arcs, dotted line in Figure 7).

Consider an arc (u, v) from u to v in T with cost $\tilde{c}_{vu}$. If (u, v) is a

- forward arc then, by construction of T, its cost must be less than or equal to the cost of the corresponding path in T; hence $\tilde{c}_{vu} \leq 0$.
- backward arc then it closes a cycle in G, which has cost at most 0. Since the cost for the arcs in T is 0 one must have $\tilde{c}_{vu} \leq 0$.
- cross arc then the cost of the path from r to v must exceed the sum of the paths from r to u and from u to v, otherwise v would be a descendant of u. Since the paths in T from r to u and from r to v have zero cost, one must have $\tilde{c}_{vu} \leq 0$.

Clearly, every c_{vu} is a multiple of g, hence (with $c_r = 0$) every c_u is a multiple of g and the Diophantine equation for the components of d_u, $\tau^T d_u = c_u$, has an infinite number of solutions which can be easily characterised. For any set of translations d_u, each satisfying the corresponding equation $\tau^T d_u = c_u$, the cost $\tilde{c}_{vu}$ on every arc satisfies $\tilde{c}_{vu} \leq 0$ so that $\tau^T \tilde{d}_{vu} \leq -g < 0$. Thus, the translations d_u and the vector τ define a valid schedule.

In general, to determine all the translations d_u which define a valid schedule for a given τ within the time cone, one would have to examine all possible assignments c_u for which $\tilde{c}_{vu} \leq 0$ (the algorithm above constructs just one such assignment).

For a vector τ *outside* the time cone there is a cycle such that $\tau^T d_{\gamma_k} > -\nu_k g$, hence the cycle contains at least one dependence vector $\tilde{d}_{vu}$ with $\tau^T \tilde{d}_{vu} \geq 0$, whatever translations are applied. For this particular time vector there is thus no set of translations d_u that defines a valid schedule.

∎

As we have seen in Section 6 the time cone corresponding to a step with uniform recurrence equations may be empty while the structure of the graph Γ indicates computable equations. Thus, a translational schedule may not always exist even though the algorithm is computable.

> Consider the 2-D filter example. Since $d_{\gamma_2} = -d_{\gamma_3}$ the time cone is empty and Theorem 7.1 implies that no translational schedule exists for this example. Yet, the algorithm is computable as was proved in Section 6. In fact the longest path through the precedence graph of the algorithm is
>
> $$\begin{array}{ccccccccccccccccc} u_{n00} & - & v_{n-1,0,0} & - & \cdots & - & v_{200} & - & v_{100} & - & w_{200} & - & \cdots & - & w_{n00} & - & \\ u_{n11} & - & v_{n-1,1,1} & - & \cdots & - & v_{211} & - & v_{111} & - & w_{211} & - & \cdots & - & w_{n11} & - & \\ \vdots & & & & & & & & & & & & & & & & \\ u_{nmm} & - & v_{n-1,m,m} & - & \cdots & - & v_{2mm} & - & v_{1mm} & - & w_{2mm} & - & \cdots & - & w_{nmm} & - & w_{n+1,m,m} \end{array}$$
>
> Hence a lower bound on the computation time is $2n(m+1) - m$ and there exist schedules attaining this bound.

Whenever a computable step of uniform recurrence equations does not possess a translational schedule, schedules at the next level of complexity, in this case linear schedules, are considered. 'Good' linear schedules are obtained from affine transformations F_u that map the domain of each variable u onto the same domain C, $\forall u : \ F_u(C_u) = C$ (after renaming all variables in the step have domains of the same size) thus making the domain C for the whole step as small as possible. Although this appears to be a only a heuristic measure with respect to total minimal computation time, we plan to provide a rigorous justification of this procedure in another paper.

The set of affine transformations that map the domain onto itself may be completely charaterised: their linear parts, for instance, have all eigenvalues on the unit circle.

> Possible transformations for a cubical domain with edges parallel to the canonical basis vectors e_1, e_2 and e_3 could be a reflection with respect to the (e_2, e_3) plane or a rotation with respect to the e_1 direction. The linear parts of these transformations are respectively
>
> $$\begin{bmatrix} -1 & & \\ & 1 & \\ & & 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & & \\ & & 1 \\ & -1 & \end{bmatrix}.$$

Since the transformations F_u induce similarity transformations on the compositions of dependence mappings $D_{\gamma_k u}$, the new $\tilde{D}_{\gamma_k u}$ are still translations but with modified translation vectors $\tilde{d}_{\gamma_k u}$. For each variable u and every elementary cycle γ_k incident on u one has $\tilde{D}_{\gamma_k u} = F_u \circ D_{\gamma_k u} \circ F_u^{-1}$ where

$$D_{\gamma_k u}(P_u) = P_u + d_{\gamma_k}, \quad F_u(P_u) = F_u P_u + d_u, \quad F_u^{-1}(P) = F_u^{-1} P - F_u^{-1} d_u.$$

Hence $\tilde{D}_{\gamma_k u}(P) = P + F_u d_{\gamma_k}$ and the modified translation vector is

$$\tilde{d}_{\gamma_k u} = F_u d_{\gamma_k}.$$

The matrices F_u are selected to render the time cone associated with the $\tilde{d}_{\gamma_k u}$ non-empty. Suppose $\Delta_u \equiv \{\tilde{d}_{\gamma_k u}\}$ is the set of all vectors associated with the elementary cycles γ_k incident on u. Since

the algorithm is computable, the time cone for each individual Δ_u is non-empty and the time cone for the union $\bigcup_u \Delta_u$ can also be made non-empty through proper choices of transformations F_u.

All matrices F_u should also leave at least one common direction invariant: there exists a vector σ such that $\forall u : F_u\sigma = \sigma$. The projections of the 'dependence vectors' $\tilde{D}_{vu}(P) - P$ along σ are independent of P, and as many directions σ as possible should be left invariant by the transformations F_u. If, in addition, the inner product between σ and all modified vectors associated with the cycles is negative, $\forall \tilde{d}_{\gamma_k u} : \tilde{d}_{\gamma_k u}\sigma \leq 0$, then there exist translation vectors d_u and time vectors τ such that the transformations $F_u(P_u) = F_u P_u + d_u$ and τ define a linear schedule for the step. Again, the proofs and the development of an algorithm for the proper choice of the F_u are deferred to a future paper.

The projections of the vectors d_{γ_k} on the (e_2, e_3)-plane define a non-empty time cone within that plane. Thus, we shall select the transformations F_u so that they leave the directions e_2 and e_3 invariant, therefore

$$F_u = \begin{bmatrix} \pm 1 & & \\ & 1 & \\ & & 1 \end{bmatrix}.$$

The first element of each matrix F_u is chosen to result in a non-empty time-cone for the vectors $\tilde{d}_{\gamma_k u}$. A possible choice would be 1, -1 and 1 for F_u, F_v and F_w, respectively, so the translation vectors for the transformations may be determined and a time vector τ selected, for instance

$$d_u = (0,0,0)^T, \quad d_v = (n,1,0)^T, \quad d_w = (-2,1,1)^T, \quad \tau = (1,n,n)^T.$$

In this schedule, the computations start at time $n+1$ and end at $n - 1 + 2n(m+1)$, requiring a total time of $2n(m+1) - 1$.

8. A View of Future Work

This last section conveys an idea of how we will deal with affine recurrence equations. We shall use an illustration and consider the scheduling of a step whose dependence mappings are more general than translations (uniform recurrences) although they still represent a very special class of affine mappings. Each composition of dependence mappings associated with a cycle $D_{\gamma_k u}(P) = D_{\gamma_k u}P + d_{\gamma_k u}$ is assumed to satisfy the following property: there exists a translation vector $\bar{d}_{\gamma_k u}$ such that $\bar{D}_{\gamma_k u}(P) \equiv D_{\gamma_k u}P + \bar{d}_{\gamma_k u}$ satisfies $\bar{D}_{\gamma_k u}(C) = C$, where C is the same domain for all the variables in the step. Uniform recurrences are obtained when the matrices $D_{\gamma_k u}$ are all equal to the identity. The dependence function $D^{(2)}_{ss}(i,j) = (j+2-i, j-1)$ in the Toeplitz factorisation algorithm provides a practical example of a non-uniform recurrence that belongs to the class under consideration.

To perform a computability analysis on such algorithms we need to study the eigenstructure of the compositions of dependence mappings around elementary cycles. First we note that a finite number of iterations of $\bar{D}_{\gamma_k u}$ results in an identity transformation.

Lemma 8.1.

$$\exists l \geq 1 : \bar{D}^l_{\gamma_k u}(P) = P \qquad \forall P \in C.$$

Proof. Since C is finite and $\bar{D}_{\gamma_k u}$ is bijective, $\bar{D}_{\gamma_k u}$ is a permutation of the elements of C. The set of permutations on C forms a finite group, with $C!$ elements. The powers of $\bar{D}_{\gamma_k u}$,

$\{I, \bar{D}_{\gamma_k u}, \bar{D}^2_{\gamma_k u}, \ldots, \}$, form a subgroup which contains at most $|C|!$ elements, hence there exists a *finite* least $l \geq 1$ such that $\bar{D}^l_{\gamma_k u}(P) = P$ for all $P \in C$.

∎

Let q be such that C is a subset of $\mathbf{Z}^q$ and is not included in a proper affine subspace of $\mathbf{Z}^q$. The rest of the discussion is presented for $q = 2$ but similar structural results may be established for an arbitrary number q of indices.

Theorem 8.1. *In* $\mathbf{Z}^2$ *either* $D_{\gamma_k u} = I$ *and* $\bar{d}_{\gamma_k u} = 0$, *or*

$$D_{\gamma_k u} = F^{-1}_{\gamma_k u} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} F_{\gamma_k u} \quad \text{and} \quad (D_{\gamma_k u} + I)\bar{d}_{\gamma_k u} = 0.$$

Proof. In this proof we simplify the notation and drop the subscripts. Because $\bar{D}$ is an affine transformation one has

$$\bar{D}^l(P) = D^l P + (D^{l-1} + \ldots + D + I)\bar{d} = P \qquad \forall P \in C$$

implying

$$(D^l - I)(P - P') = 0 \qquad \forall P, P' \in C.$$

The ability to select q linearly independent vectors $P_i - P'_i$ where $P_i, P'_i \in C$ implies that $D^l = I$. from which $(D^{l-1} + \ldots + D + I)\bar{d} = 0$ follows (moreover $\det D^l = 1$ so D is unimodular). When $q = 2$ the Schur decomposition of D can be written as

$$D = \bar{F}^{-1} \begin{bmatrix} \lambda_1 & \mu \\ 0 & \lambda_2 \end{bmatrix} \bar{F}.$$

Suppose $\lambda_1 \neq 1$ and $\lambda_2 \neq 1$, then

$$(D^{l-1} + \ldots + D + I) = (D - I)^{-1}(D^l - I) = 0,$$

hence $D^l(P) = P$ in contradiction to computability. Consequently, at least one of the eigenvalues must equal one, say $\lambda_1 = 1$ and, since D is unimodular, $\lambda_2 = \pm 1$.

$\lambda_2 = 1$:

$$D^l = \bar{F}^{-1} \begin{bmatrix} 1 & l\mu \\ 0 & 1 \end{bmatrix} \bar{F} = I$$

implies

$$\begin{bmatrix} 1 & l\mu \\ 0 & 1 \end{bmatrix} = I.$$

hence $\mu = 0$ and $D = I$. Thus $l = 1$ and $\bar{d} = 0$.

$\lambda_2 = -1$:

$$D = F^{-1} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} F, \quad \text{where} \quad F = \begin{bmatrix} 1 & \mu/2 \\ 0 & 1 \end{bmatrix} \bar{F},$$

hence

$$D^2 = F^{-1} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}^2 F = I.$$

Thus $l = 2$ and $(D + I)\bar{d} = 0$.

∎

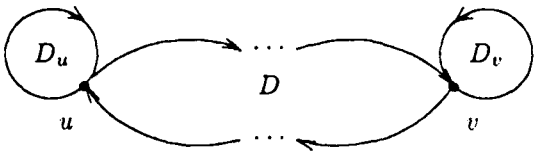


Figure 8: Illustration of the Proof of Theorem 8.2.

In a given cycle γ_k, the linear part of the mapping $D_{\gamma_k u}$ starting at a variable u is a product of the linear parts of the individual dependence mappings in γ_k. The linear parts of the mappings $D_{\gamma_k u}$ and $D_{\gamma_k v}$ starting at variables u and v, respectively, in the same cycle γ_k are cyclic permutations of each other. Thus the eigenvalues of $D_{\gamma_k u}$ depend only on the cycle γ_k [10] and are independent of the starting variable u. Consequently, when $q = 2$ each elementary cycle can be associated with one of two types of dependence mappings: those that are just translations or those possessing as eigenvalues 1 and -1. Note that the type of mapping for a given cycle is independent of its starting variable.

For each composition of dependence mappings $D(P) = DP + d$ associated with an arbitrary cycle starting at variable u there exists a vector $\bar{d}$ for which $\bar{D}(P) = DP + \bar{d}$ satisfies $\bar{D}(C) = C$. The arguments in Theorem 8.1 still apply and the linear part D must either be the identity or have eigenvalues 1 and -1. The eigenvalues associated with the cycle will be shown to be only functions of the number of times each of its elementary cycles is traversed. The fact that the eigenvalues are independent of the starting variable constitutes a non-obvious generalisation of the above result for elementary cycles and represents a step towards an efficient verification of computability.

Our key idea consists of performing index changes F_* for the variables so that the linear parts of all dependence mappings in the step share the same set of eigenvectors. These transformations F_* are assigned according the following simple rule: construct a spanning tree T for the step and go through T assigning $F_r = I$ for the root r and the other F_u such that $\tilde{D}_{vu} = I$ for each arc (u, v) in the tree. Thus the changes of index ensure that the linear part of the dependence mappings *corresponding to the arcs in the spanning tree* is equal to the identity. We shall now prove that, after this conversion, the linear parts of the dependence mappings are either the identity or else are equal to the *same* matrix D with eigenvalues 1 and -1.

Lemma 8.2. *In $\mathbf{Z}^2$ two cycles with non-identity linear part and incident on the same node must have the same linear part D.*

Proof. Let the dependence mappings corresponding to these two cycles be $D_1(P)$ and $D_2(P)$. From the extension of Theorem 8.1 to arbitrary cycles we know that $det(D_1) = det(D_2) = -1$. The linear part $D_2 D_1$ of the composition of these two dependence mappings $D_2 \circ D_1$ must have determinant one hence it must be identity, using again the extension of Theorem 8.1 to arbitrary cycles. Therefore $D_2 = D_1^{-1} = D_1$ since $D_1^2 = I$.

∎

Theorem 8.2. *In $\mathbf{Z}^2$ if two cycles in a strongly connected component have non-identity linear part then they have the same linear part D.*

Proof. Let u and v, respectively, be nodes traversed by the two cycles, see Figure 8; u and v are assumed to be distinct otherwise the previous lemma applies directly. Let D_u and D_v be the associated compositions of dependence mappings. Since they are part of a strongly connected component u and v are also nodes of another cycle. From Lemma 8.2 the linear parts of the compositions associated with this cycle must be the same, say D, independently of the starting

node. Then either $D = I$ or $D \neq I$ and then, by Lemma 8.2, $D = D_u$ and $D = D_v$, implying $D_u = D_v$. If $D = I$ then there is one path, say D_{uv}, between u and v that is in the spanning tree and $D = D_{uv}D_{vu}$ (note that both D_{uv} and D_{vu} may be compositions of individual D_e). Since the change of index transformations have been applied, all arcs on the path D_{uv} have been assigned identity matrices, so that $D_{uv} = I$, hence $D_{vu} = I$. Since $D_{uv} = D_{vu} = I$ the mapping $D_u D_{uv} D_v D_{vu} : C_u \rightarrow C_u$ is $D_u D_v$, and since $\det(D_u D_v) = 1$ it follows that $D_u D_v = I$. Thus $D_u = D_v$.

∎

Since the cycles with non-identity linear parts have the same linear part D, the change of index transformations ensure that the linear part of each dependence mapping automatically assumes the value D or I. Let

$$D = F^{-1} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} F.$$

If the change of index $F_u \equiv F$ is applied to every variable, the dependence mappings are of the form

$$D_{vu}(P) = \begin{bmatrix} 1 & \\ & 1 \end{bmatrix} P + d_{vu}, \quad \text{or} \quad D_{vu}(P) = \begin{bmatrix} 1 & \\ & -1 \end{bmatrix} P + d_{vu}.$$

It is now clear that the eigenvalues associated with an arbitrary cycle are only a function of the number of times each of its elementary cycles is traversed. Equipped with this result, we could now carry out a computability analysis similar to the one for uniform recurrence equations.

The above results are also of importance for the scheduling of such recurrence equations. In order to find schedules that are as simple as possible one could attempt one further index transformation rendering the dependence vectors constant (independent of P). On each variable a 'folding transformation' may be performed that consists of mapping one half of the domain onto the other half by folding it along an axis parallel to the e_1-direction. The folding is accompanied by a renaming of the variables associated with the folded half of the domain so that a variable is computed only once for each point in the final domain. The dependence vectors become independent of P (except for P 'close' to the folding axis). As in the case of uniform recurrence equations, simple schedules associated with a folded domain may not always exist even though the algorithm is computable. One must then give up constant dependences and resort to more complex affine schedules.

Acknowledgement

We wish to thank Yiwan Wong for helpful discussions.

References

[1] Delosme, J.-M. and Ipsen, I.C.F., *Efficient Systolic Arrays for the Solution of Toeplitz Systems : An Illustration of a Methodology for the Construction of Systolic Architectures in VLSI.* Research Report 370, Dept Computer Science, Yale University, 1985.

[2] ————, An Illustration of a Methodology for the Construction of Efficient Systolic Architectures in VLSI, *Proc. Second Int. Symposium on VLSI Technology, Systems and Applications*, Taipei, Taiwan, 1985, pp. 268-73.

[3] Karp, R.M. and Miller, R.E., *Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing*, SIAM J. Appl. Math, 14 (1966), pp. 1390-411.

[4] Karp, R.M., Miller, R.E. and Winograd, S., *The Organization of Computations for Uniform Recurrence Equations*, JACM, 14 (1967), pp. 563-90.

[5] Kuck, D.J., *The Structure of Computers and Computations*, John Wiley & Sons, 1978.

[6] Moldovan, D.I., *On the Design of Algorithms for VLSI Systolic Arrays*, IEEE Trans. Comp., C-31 (1982), pp. 1121-6.

[7] Quinton, P., Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations, *Proc. 11th Annual Intern. Symp. Computer Architecture*, IEEE, 1984, pp. 208-14.

[8] Rao, S.K., *Regular Iterative Algorithms and their Implementations on Processor Arrays*, Ph.D. Thesis, Dept of Electrical Engineering, Stanford University, 1985.

[9] Tarjan, R., *Depth-First Search and and Linear Graph Algorithms*, SIAM J. Comput., 1 (1972), pp. 146-60.

[10] Wilkinson, J.H., *The Algebraic Eigenvalue Problem*, Clarenson Press, Oxford, 1965.

END

DTIC

8-86